Chapter Four:
Database Design
Using Normalization
Chapter Objectives

• To design updatable databases to store data received from another source
• To use SQL to access table structure
• To understand the advantages and disadvantages of normalization
• To understand denormalization 去正規化
• To design read-only databases to store data from updateable databases
Chapter Objectives

• To recognize and be able to correct common design problems:
  – The multivalue, multicolumn problem
  – The inconsistent values problem
  – The missing values problem
  – The general-purpose remarks column problem
Chapter Premise

• We have received one or more tables of existing data.
• The data is to be stored in a new database.
• QUESTION: Should the data be stored as received, or should it be transformed for storage?
How Many Tables?

SKU_DATA (SKU, SKU_Description, Buyer)
BUYER (Buyer, Department)

Where SKU_DATA.Buyer must exist in BUYER.Buyer

Should we store these two tables as they are, or should we combine them into one table in our new database?
### Assessing Table Structure

- Count rows and examine columns
- Examine data values and interview users to determine:
  - Multivalued dependencies
  - Functional dependencies
  - Candidate keys
  - Primary keys
  - Foreign keys
- Assess validity of assumed referential integrity constraints
Counting Rows in a Table

• To count the number of rows in a table use the SQL COUNT(*) built-in function:

```
SELECT COUNT(*) AS NumRows
FROM SKU_DATA;
```
Examining the Columns

• To determine the number and type of columns in a table, use an SQL SELECT statement.

• To limit the number of rows retrieved, use the SQL TOP \{NumberOfRows\} expression:

  ```sql
  SELECT TOP (10) *
  FROM SKU_DATA;
  ```
Checking Validity of Assumed Referential Integrity Constraints I

• Given two tables with an assumed foreign key constraint:

  SKU_DATA  (SKU, SKU_Description, Buyer)
  BUYER     (Buyer, Department)

  Where SKU_DATA.Buyer must exist in BUYER.Buyer
Checking Validity of Assumed Referential Integrity Constraints II

• To find any foreign key values that violate the foreign key constraint:

```sql
SELECT Buyer
FROM SKU_DATA
WHERE Buyer NOT IN
    (SELECT SKU_DATA.Buyer
     FROM SKU_DATA, BUYER
     WHERE SKU_DATA.BUYER = BUYER.Buyer);
```
Type of Database

- Updateable database, or read-only database?
- If updateable database, we normally want tables in BCNF.
- If read-only database, we may not use BCNF tables.
Designing
Updatable Databases
Normalization: Advantages and Disadvantages

- Advantages
  - Eliminate modification anomalies
  - Reduce duplicated data
    - Eliminate data integrity problems
    - Save file space
- Disadvantages
  - More complicated SQL required for multitable subqueries and joins
  - Extra work for DBMS can mean slower applications
## Non-Normalized Table: EQUIPMENT_REPAIR

<table>
<thead>
<tr>
<th>ItemNumber</th>
<th>Equipment Type</th>
<th>AcquisitionCost</th>
<th>RepairNumber</th>
<th>RepairDate</th>
<th>RepairCost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Drill Press</td>
<td>3500.00</td>
<td>2000</td>
<td>2011-05-05 ...</td>
<td>375.00</td>
</tr>
<tr>
<td>2</td>
<td>Lathe</td>
<td>4750.00</td>
<td>2100</td>
<td>2011-05-07 ...</td>
<td>255.00</td>
</tr>
<tr>
<td>3</td>
<td>Drill Press</td>
<td>3500.00</td>
<td>2200</td>
<td>2011-06-19 ...</td>
<td>178.00</td>
</tr>
<tr>
<td>4</td>
<td>Mill</td>
<td>27300.00</td>
<td>2300</td>
<td>2011-06-19 ...</td>
<td>1875.00</td>
</tr>
<tr>
<td>5</td>
<td>Drill Press</td>
<td>3500.00</td>
<td>2400</td>
<td>2011-07-05 ...</td>
<td>0.00</td>
</tr>
<tr>
<td>6</td>
<td>Drill Press</td>
<td>3500.00</td>
<td>2500</td>
<td>2011-08-17 ...</td>
<td>275.00</td>
</tr>
</tbody>
</table>
### EQUIPMENT_ITEM

<table>
<thead>
<tr>
<th>ItemNumber</th>
<th>Equipment Type</th>
<th>AcquisitionCost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Drill Press</td>
<td>3500.00</td>
</tr>
<tr>
<td>2</td>
<td>Lathe</td>
<td>4750.00</td>
</tr>
<tr>
<td>3</td>
<td>Mill</td>
<td>27300.00</td>
</tr>
</tbody>
</table>

### REPAIR

<table>
<thead>
<tr>
<th>RepairNumber</th>
<th>ItemNumber</th>
<th>RepairDate</th>
<th>RepairCost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>2011-05-05</td>
<td>375.00</td>
</tr>
<tr>
<td>2</td>
<td>200</td>
<td>2011-05-07</td>
<td>255.00</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>2011-06-19</td>
<td>178.00</td>
</tr>
<tr>
<td>4</td>
<td>300</td>
<td>2011-06-19</td>
<td>1875.00</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>2011-07-05</td>
<td>0.00</td>
</tr>
<tr>
<td>6</td>
<td>100</td>
<td>2011-08-17</td>
<td>275.00</td>
</tr>
</tbody>
</table>
To copy data from one table to another, use the SQL command `INSERT INTO TableName` command:

```
INSERT INTO EQUIPMENT_ITEM
    SELECT DISTINCT ItemNumber, EquipmentType, AcquisitionCost
    FROM EQUIPMENT_REPAIR;
```

```
INSERT INTO REPAIR
    SELECT RepairNumber, ItemNumber, RepairDate, RepairCost
    FROM EQUIPMENT_REPAIR;
```
Choosing Not To Use BCNF

- BCNF is used to control anomalies from functional dependencies.
- There are times when BCNF is not desirable.
- The classic example is ZIP codes:
  - ZIP codes almost never change.
  - Any anomalies are likely to be caught by normal business practices.
  - Not having to use SQL to join data in two tables will speed up application processing.
Multivalued Dependencies

• Anomalies from multivalued dependencies are very problematic.
• *Always* place the columns of a multivalued dependency into a separate table (4NF).
Designing Read-Only Databases
Read-Only Databases

- **Read-only databases** are nonoperational databases using data extracted from operational databases.
- They are used for querying, reporting, and data mining applications.
- They are never updated (in the operational database sense—they may have new data imported from time to time).
Denormalization

- For read-only databases, normalization is seldom an advantage.
  - Application processing speed is more important.
- **Denormalization** is the joining of the data in normalized tables prior to storing the data.
- The data is then stored in nonnormalized tables.
Normalized Tables

<table>
<thead>
<tr>
<th>STUDENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>StudentID</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ACTIVITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PAYMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>StudentID</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
</tbody>
</table>
Denormalizing the Data

```
INSERT INTO STUDENT_ACTIVITY_PAYMENT_DATA
SELECT STUDENT.StudentID, StudentName,
       ACTIVITY.Activity,
       ActivityFee, AmountPaid
FROM STUDENT, PAYMENT, ACTIVITY
WHERE STUDENT.StudentID = PAYMENT.StudentID
AND PAYMENT.Activity = ACTIVITY.Activity;
```
Customized Tables I

• Read-only databases are often designed with many copies of the same data, but with each copy customized for a specific application.

• Consider the PRODUCT table:

- SKU (Primary Key)
- PartNumber (Candidate key)
- SKU_Description (Candidate key)
- VendorNumber
- VendorName
- VendorContact_1
- VendorContact_2
- VendorStreet
- VendorCity
- VendorState
- VendorZip
- QuantitySoldPastYear
- QuantitySoldPastQuarter
- QuantitySoldPastMonth
- DetailPicture
- ThumbNailPicture
- MarketingShortDescription
- MarketingLongDescription
- PartColor
- UnitsCode
- BinNumber
- ProductionKeyCode
Customized Tables II

PRODUCT_PURCHASING (SKU, SKU_Description, VendorNumber, VendorName, VendorContact_1, VendorContact_2, VendorStreet, VendorCity, VendorState, VendorZip)

PRODUCT_USAGE (SKU, SKU_Description, QuantitySoldPastYear, QuantitySoldPastQuarter, QuantitySoldPastMonth)

PRODUCT_WEB (SKU, DetailPicture, ThumbnailPicture, MarketingShortDescription, MarketingLongDescription, PartColor)

PRODUCT_INVENTORY (SKU, PartNumber, SKU_Description, UnitsCode, BinNumber, ProductionKeyCode)
Common Design Problems

- Multivalue, Multicolumn Problem
- Inconsistent Values
- Missing Values
- General-Purpose Remarks Column
The **multivalue, multicolumn problem** occurs when multiple values of an attribute are stored in more than one column:

```
EMPLOYEE (EmployeeNumber, EmployeeLastName, EmployeeLastName, Email, Auto1_LicenseNumber, Auto2_LicenseNumber, Auto3_LicenseNumber)
```

- This is another form of a multivalued dependency.
- Solution = like the 4NF solution for multivalued dependencies, use a separate table to store the multiple values.
Inconsistent Values I

- **Inconsistent values** occur when different users, or different data sources, use slightly different forms of the same data value:
  - Different codings:
    - SKU_Description = 'Corn, Large Can'
    - SKU_Description = 'Can, Corn, Large'
    - SKU_Description = 'Large Can Corn'
  - Different spellings:
    - Coffee, Cofee, Coffeee
Inconsistent Values II

• Particularly problematic are primary or foreign key values.
• To detect:
  – Use referential integrity check already discussed for checking keys.
  – Use the SQL GROUP BY clause on suspected columns.
Inconsistent Values III

\[
\text{SELECT} \quad \text{SKU\_Description, COUNT(*) AS NameCount} \\
\text{FROM} \quad \text{SKU\_DATA} \\
\text{GROUP BY} \quad \text{SKU\_Description};
\]
Missing Values

• A **missing value** or **null value** is a value that has never been provided.
Null Values

• Null values are ambiguous:
  – May indicate that a value is inappropriate;
    • DateOfBirth is inappropriate for a male.
  – May indicate that a value is appropriate but unknown;
    • DateOfBirth is appropriate for a female, but may be unknown.
  – May indicate that a value is appropriate and known, but has never been entered;
    • DateOfBirth is appropriate for a female, and may be known but no one has recorded it in the database.
Checking for Null Values

• Use the SQL keyword IS NULL to check for null values:

```
SELECT COUNT(*) AS QuantityNullCount
FROM ORDER_ITEM
WHERE Quantity IS NULL;
```
A *general-purpose remarks column* is a column with a name such as:
- Remarks
- Comments
- Notes

It often contains important data stored in an inconsistent, verbal, and verbose way.
- A typical use is to store data on a customer’s interests.

Such a column may:
- Be used inconsistently
- Hold multiple data items
End of Presentation:
Chapter Four